INTERNET-DRAFT
Expires: March 1, 2000

Greg Hudson
ghudson@mit.edu
MIT

Simple Protocol Application Data Encoding

draft-hudson-spade-03.txt

1. Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Please send comments to ghudson@mit.edu.

2. Abstract

This document describes a simple scheme for encoding network protocol data and a simple notation for describing protocol data elements. All encodings are self-terminating (you know when you've reached the end) and assume that the decoder knows what type of protocol element it is expecting.

3. Encoding

This encoding scheme uses the ASCII translation of characters into bytes except when otherwise noted. Protocol elements are encoded as follows:

> A byte: a byte encodes itself. So "a" encodes as "a" and so forth.
>
> An integer: a sequence of decimal digits followed by a colon. For instance, the number 27 encodes as "27:". Negative integers are preceded by a minus sign, so -27 encodes as "-27:". Excess leading zeroes are not allowed, and zero must be encoded as "0:" (not as "-0:"). Thus each integer has a single encoding.

A symbol: a sequence of letters, numbers, and dashes, beginning with a letter, followed by a colon. Case is significant. For instance, the symbol "foo" encodes as "foo:".

A list of <type>: an integer giving the number of elements in the list, followed by the elements of the list. For instance, the list of numbers "1", "2", and "3" encodes as "3:1:2:3:".

A structure: a collection of dissimilar elements can simply be concatenated together. For instance, a structure containing the number 3 and the list of bytes "ab" encodes as "3:2:ab".

A union value: a symbol giving the type of element, an integer giving the length of the encoding of the element's data, and the data itself. For instance, an element of type "foo" with the same data as in the structure example above would be encoded as "foo:6:3:2:ab". If there is no data to be encoded, a data length of 0 should be given, e.g. "bar:0:".

Of these types, the union value is the most complicated. It should be used where a protocol needs extensibility, such as for a command set, and avoided elsewhere if possible.

Following is an ABNF (RFC 2234) for the wire encoding. The grammar is highly ambiguous, since the encoding assumes that the decoder has auxiliary type information:

```
element         = byte / integer / symbol / list / struct
                  / union-val
byte            = OCTET
unsigned        = ("0" / PDIGIT *DIGIT) ":"
integer         = unsigned / ("-" PDIGIT *DIGIT ":")
symbol          = ALPHA *(ALPHA / DIGIT / "-") ":"
list            = unsigned *element
                  ; unsigned determines the number of elements
                  ; all elements must be of same type
struct          = 1*element
                  ; number, type, and order depend on type
union-val       = symbol unsigned [element]
                  ; unsigned is the length of the element
                  ; unsigned is 0 if element is omitted
PDIGIT          = %x31-39
                  ; 1-9
```

4. Notation

This notation gives a scheme for describing protocol element types and giving them names for the purpose of semantic descriptions.

A variable declaration associates a name to be used in semantic

descriptions with a type. Variable names are valid symbols beginning with a lowercase letter. Variable declarations end with a line break, and are written as follows:

A byte: "Byte <name>"
An integer: "Integer <name>"
A symbol: "Symbol <name>"
A list of <type>: "List[<type>] <name>"
A structure named <structurename>: "<structurename> <name>"
A union named <unionname>: "<unionname> <name>"

As a notational short-hand, "String" may be used as a synonym for "List[Byte]".

Structure and union names are valid symbols beginning with a capital letter. A structure definition is written as:

```
structure <structurename> {
        <variable declaration>
        .
        .
        .
}
```

Unions are defined as:

```
union <unionname> {
        <symbol>: <variable declaration>
        .
        .
        .
}
```

As a special case, if there is no data for a particular union tag, "Null" can be written in place of a variable declaration.

Here is an example of two structure definitions which might be used to describe a mail message:

```
structure Header {
        String name
        String value
}

structure Message {
        List[Header] headers
        String body
}
```

Here is an example of a union definition which might be used together with the above structure definitions to describe a command set:

```
union Command {
        send: Message m
        help: Null
        quit: Null
}
```

A quit command would be encoded as "quit:0:". If I have a message with two headers, one with name "From" and value "Greg" and another with name "To" and value "Bob", and the message body is "Test", then I would encode a command to send this message as:

```
send:29:2:4:From4:Greg2:To3:Bob4:Test
```

5. Rationale

The primary goal of this encoding scheme is simplicity. Protocol implementors should not have to read a book to understand how data is encoded. For want of a simple encoding scheme, IETF protocols have been turning to ASN.1's basic encoding rules, which are highly complicated and which have presented a barrier to implementation in practice.

Two secondary goals of this encoding scheme are human readability and space efficiency. These goals are of course at odds; integers could be encoded more compactly by using more than ten values per byte, for instance, at the expense of making it more difficult to examine ASCII translations of protocol data.

The tagged union encoding provides easy extensibility in most protocols. A protocol can find the end of the encoding of a tagged union element even if it doesn't know the data types for the individual tags.

This encoding does not include a length field for structures or an overall length field for lists. Thus, it is impossible to skip to the end of a structure or list without decoding it. This decision was a tradeoff; it simplifies encoding and uses space more efficiently in return for making certain decoding situations more complicated.

6. Comparisons

This section compares SPADE to several previous wire encodings, including XDR (RFC 1832), CDR (CORBA/IIOP 2.3 section 15.3), ABNF (RFC 2234), ASN.1 (ITU-T X.680-X.691), and XML (W3 REC-XML).

XDR is similar in philosophy to SPADE: data can only be decoded with auxiliary type information, and simple schemes are used to encode structures and lists. However, XDR made tradeoffs which make the encoding more complex and less general than SPADE. Here are the major differences:

* XDR uses a fixed-length binary encoding for integers

(usually 32-bit) and enforces four-byte alignment on the encoding of all data types, so that a C implementation on most platforms can byte-swap an integer encoding and cast it to the appropriate type. As a result, there are two different sizes of integers, integers are constrained by any XDR-using protocol to a certain range, integer encodings are not human-readable in an ASCII protocol trace, and padding is required for the encoding and decoding of strings.

* XDR needlessly distinguishes between "opaque data" (bytes) and "strings" (ASCII characters).

* XDR's "discriminated union" does not provide the length field needed to extend the union in later revisions of a protocol. As a result, unions have less overhead, but it is harder to make a protocol extensible.

* XDR does not provide symbols; instead, it provides enumerations which map symbols to numbers. In most cases, such mappings are unnecessary and a debugging hindrance. The discriminant of an XDR union must be a number.

* XDR has more types: it provides floating point numbers, fixed-length lists, and optional data, and it distinguishes between signed and unsigned integers.

CDR is similar to XDR in all of the respects noted above. It is slightly more complicated than XDR: it provides a wider variety of integral types, has different alignment constraints for different types, and allows integral types to be encoded in either little-endian or big-endian form (XDR always uses big-endian).

ABNF is not really a wire encoding scheme at all; it is a scheme for describing syntaxes. It can be used to describe any wire encoding, but does not in itself nail down how integers, strings, lists, etc. should be encoded. As a notational device, it is not geared towards describing data types; the description of a data type would have to be intrinsically linked to its encoding, which is undesirable, and ABNF has no notion of counting, so it cannot link the length field of a list with the number of list elements provided.

ASN.1 is a very large and complicated specification. Even third-party attempts to describe the ASN.1 basic and distinguished encoding rules tend to be long and difficult to understand. Some differences between ASN.1 with the Distinguished Encoding Rules (DER) and SPADE are:

* The DER provide explicit type information in the encoding. As a result, a DER decoder can turn a wire encoding into a data structure with no auxiliary type information, but the encoding is space-inefficient and the encoding scheme becomes much more complex. Someone writing an encoder by

hand for an ASN.1-using protocol will be constantly adding in magic numbers which are always the same.

* The DER provide an overall length field for lists ("SEQUENCE OF"), but not an element count. So a decoder can easily skip over a list without parsing the elements, but it cannot allocate memory for the list elements until it has decoded the list.

* The DER provide an overall length field for structures ("SEQUENCE"), making it easy to skip over structures without decoding their contents. The cost is a less space-efficient protocol.

* The DER rely heavily on bit-packing to encode type tags and lengths, resulting in a very complex wire encoding which is nonetheless not space-efficient.

* ASN.1 needlessly distingishes between "OCTET STRING", "IA5String" (ASCII), "PrintableString" (restricted ASCII), and "T61String" (extended ASCII). A wire encoding does not need to get into the interpretation of octets as characters.

XML takes a different view of encoding protocol data than SPADE, ASN.1, or XDR. The only "primitive data type" is character data, which is arranged into a tree of elements according to a Document Type Definition (DTD). Protocol data is encoded in a markup language instead of being packed precisely according to rigid data types. Some resulting differences between XML and SPADE are:

* An XML encoding will be much more verbose than a SPADE encoding (or even an ASN.1 DER encoding in most cases), due to the addition of element names. Of course, this makes it easier for a human reader of a protocol dump to guess the meaning of each data element.

* Because XML uses markup rather than advance-length encoding, an encoder or decoder has more quoting issues to deal with. This and other facets of XML make XML a much more complicated specification than SPADE.

* An XML DTD is generally not understandable to a reader not skilled in XML, whereas a SPADE or XDR or ASN.1 data type generally is.

* Hand-coding an encoder or decoder for an XML-using protocol is probably a lost cause.

7. Security Considerations

For maximum generality, this encoding scheme places no limits on the length of any data type. This could lead to denial of service attacks

against implementations of protocols using this encoding ("here follows a string of length two gazillion"). It does not seem appropriate to choose limits in the wire encoding to prevent this sort of attack, so guarding against these attacks will have to be the responsibility of particular protocols or their implementations.

8. Acknowledgements

Thanks to Elliot Schwartz for suggesting the name.

Thanks to Chris Newman for providing the basis for the ABNF for the wire encoding, and other useful suggestions.